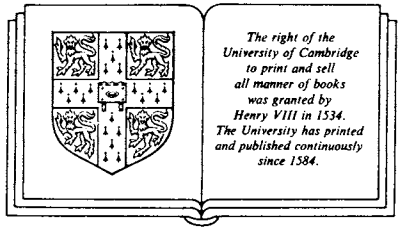
26 Cambridge Computer Science Texts

Concurrent Programming

C. R. Snow
University of Newcastle upon Tyne



Cambridge University Press

Cambridge
New York Port Chester Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Victoria 3166, Australia

First published 1992

Printed in Great Britain at the University Press, Cambridge

Library of Congress cataloguing in publication data available

British Library cataloguing in publication data available

ISBN 0 521 32796 2 hardback
ISBN 0 521 33993 6 paperback

Contents

1
Introduction to Concurrency

Concurrency has been with us for a long time. The idea of different tasks being carried out at the same time, in order to achieve a particular end result more quickly, has been with us from time immemorial. Sometimes the tasks may be regarded as independent of one another. Two gardeners, one planting potatoes and the other cutting the lawn (provided the potatoes are not to be planted on the lawn!) will complete the two tasks in the time it takes to do just one of them. Sometimes the tasks are dependent upon each other, as in a team activity such as is found in a well-run hospital operating theatre. Here, each member of the team has to co-operate fully with the other members, but each member has his/her own well-defined task to carry out.

Concurrency has also been present in computers for almost as long as computers themselves have existed. Early on in the development of the electronic digital computer it was realised that there was an enormous discrepancy in the speeds of operation of electro-mechanical peripheral devices and the purely electronic central processing unit. The logical resolution of this discrepancy was to allow the peripheral device to operate independently of the central processor, making it feasible for the processor to make productive use of the time that the peripheral device is operating, rather than have to wait until a slow operation has been completed. Over the years, of course, this separation of tasks between different pieces of hardware has been refined to the point where peripherals are sometimes controlled by a dedicated processor which can have the same degree of "intelligence" as the central processor itself.

Even in the case of the two gardeners, where the task that each gardener was given could be considered to be independent of the other, there must be some way in which the two tasks may be initiated. We

may imagine that both of the gardeners were originally given their respective tasks by the head gardener who, in consultation with the owner of the garden, determines which tasks need to be done, and who allocates tasks to his under-gardeners. Presumably also, each gardener will report back to the head gardener when he has finished his task, or maybe the head gardener has to enquire continually of his underlings whether they have finished their assigned tasks.

Suppose, however, that our two gardeners were asked to carry out tasks, both of which required the use of the same implement. We could imagine that the lawn-mowing gardener requires a rake to clear some debris from the lawn prior to mowing it, while the potato planter also requires a rake to prepare the potato bed before planting. If the household possessed only a single rake, then one or other gardener might have to wait until the other had finished using it before being able to complete his own task.

This analogy serves to illustrate the ways in which peripheral devices may interact with central processors in computers. Clearly if we are asking for a simple operation to take place, such as a line printer skipping to the top of the next page, or a magnetic tape rewinding, it suffices for the Central Processing Unit (c.p.u.) to initiate the operation and then get on with its own work until the time when either the peripheral device informs the c.p.u. that it has finished (i.e. by an interrupt), or the c.p.u. discovers by (possibly repeated) inspection that the operation is complete (i.e. by polling). Alternatively, the peripheral device may have been asked to read a value from an external medium and place it in a particular memory location. At the same time the processor, which is proceeding in its own time with its own task, may also wish to access the same memory location. Under these circumstances, we would hope that one of the operations would be delayed until the memory location was no longer being used by the other.

1.1 Reasons for Concurrency

Concurrent programming as a discipline has been stimulated primarily by two developments. The first is the concurrency which had been introduced in the hardware, and concurrent programming could be seen as an attempt to generalise the notion of tasks being allowed to proceed largely independently of each other, in order to mimic the

relationship between the various hardware components. In particular, the control of a specific hardware component is often a complex task requiring considerable ingenuity on the part of the programmer to produce a software driver for that component. If a way could be found by which those aspects of the driver which are concerned with the concurrent activity of the device might be separated off from other parts in the system, the task is eased tremendously. If concurrent programming is employed, then the programmer can concern himself with the sequential aspects of the device driver, and only later must he face the problem of the interactions of the driver with other components within the system. In addition, any such interactions will be handled in a uniform and (hopefully) well-understood way, so that (device-) specific concurrency problems are avoided.

The second development which leads directly to a consideration of the use of concurrent programming is a rationalisation and extension of the desire to provide an operating system which would allow more than one user to make use of a particular computer at a time. Early time-sharing systems which permitted the simultaneous use of a computer by a number of users often had no means whereby those users (or their programs) could communicate with one another. Any communication which was possible was done at the operating system kernel level, and this was usually a single monolithic program which halted all the user tasks while it was active. Users of such systems were not generally concerned with communicating with each other, and the only form of resource sharing that they required was in the form of competition for resources owned by the operating system. Later systems which came along began to require the possibility of users sharing information amongst themselves, where such information was not necessarily under the control of the operating system. Data could frequently be passed from one user program to another much more conveniently than using a cumbersome mechanism of asking one program to write data into a file to be read by the other program.

The introduction of concurrent programming techniques was also recognised to be a useful tool in providing additional structure to a program. We remarked earlier that the task of constructing a device driver is considerably simplified if the concurrency aspects can be set aside, and then added in a controlled way. This is very similar in

concept to some of the well-established techniques of structured programming, in which the communication between the various parts of a (sequential) program is strictly controlled, for example through the use of procedures and parameter lists. Structured programming also leaves open the possiblity of delaying the coding of various parts of the program until a later, more convenient time, allowing the writer of the program to concentrate on the specific task on hand.

In a similar way, the writer of a concurrent program may write a sequential program, leaving aside the questions of the interaction with other concurrently active components until the sequential program is complete and, possibly, partially tested. We suspect that unstructured parallelism in programming would be even more difficult to manage than an unstructured sequential program unless we were able to break down the concurrency into manageable sub-units. Concurrent programming may therefore be regarded as another manifestation of the "divide and conquer" rule of program construction. Such methodologies are also useful as a way of making programs more readable and therefore more maintainable.

1.2 Examples of Concurrency

There are many useful examples of concurrency in everyday life, in addition to the example of the two gardeners mentioned above. Any large project, such as the building of a house, will require some work to go on in parallel with other work. In principle, a project like building a house does not require any concurrent activity, but it is a desirable feature of such a project in that the whole task can be completed in a shorter time by allowing various sub-tasks to be carried out concurrently. There is no reason why the painter cannot paint the outside of the house (weather permitting!), while the plasterer is busy in the upstairs rooms, and the joiner is fitting the kitchen units downstairs. There are however some constraints on the concurrency which is possible. The bricklayer would normally have to wait until the foundations of the house had been laid before he could begin the task of building the walls. The various tasks involved in such a project can usually be regarded as independent of one another, but the scheduling of the tasks is constrained by notions of "task A must be completed before task B can begin".

A second example is that of a railway network. A number of trains may be making journeys within the network, and by contrast with the previous example, when they start and when they end is generally independent of most of the other journeys. Where the journeys do interact though, is at places where routes cross, or use common sections of track for parts of the journeys. We can in this example regard the movement of the trains as programs in execution, and sections of track as the resources which these programs may or may not have to share with other programs.

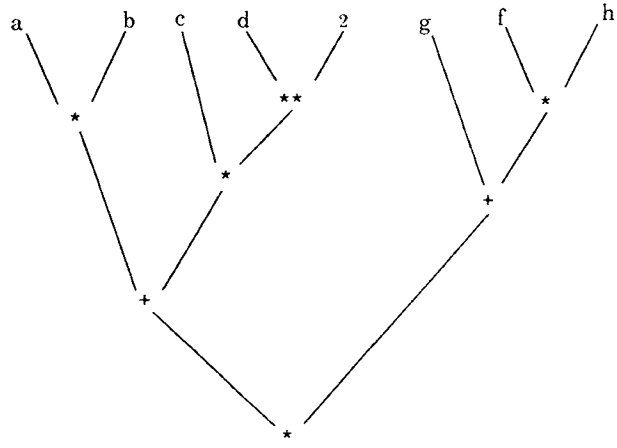
In some cases, the concurrency is inherent in the situation being considered. Any complex machine, or large plant such as a power station, chemical works or oil refinery, consists of identifiable components which have to be continuously interacting with other components. In a quality-controlled environment, for example, the product of the manufacturing component of the system is subjected to certain test procedures, which in turn provide information which may modify the way in which the manufacturing component behaves. Clearly these two components need to be constantly active and in constant communication with each other for the whole system to work properly.

An example of concurrency directly related to computing and programming can be seen by considering the evaluation of an arithmetic expression. Suppose we wish to evaluate the expression:

$$(a*b + c*d**2)*(g + f*h)$$

We assume that the identifiers a, b, c, etc. have values associated with them, and that the priority rules for evaluation of the expression are as would be expected, i.e. exponentiation first, multiplication second and addition last, modified in the usual way by the inclusion of parentheses. A tree may be drawn (figure 1.1) showing the interdependencies of the sub-expressions within the whole expression, and we may use this tree to identify possible concurrency within the evaluation. Three concurrent evaluations of sub-expressions can begin at once, namely, $a*b$, $d**2$ and $f*h$. When the second and third of these are finished, the multiplication by c, and the addition of g (respectively) can take place, also in parallel. It is only after $c*d**2$

Figure 1.1



has been evaluated that the sub-expression $a*b$ can be added, and then finally the evaluation of the whole expression can be completed.

It is in the field of operating systems where concurrent programming has been most fruitfully employed. A time-sharing operating system, by its very nature, is required to manage several different tasks in parallel, but even if the system is only providing services to a single user it will be responsible for managing all the peripheral devices as well as servicing the user(s). If the concurrent programming facilities can also be offered to the user, then the flexibility of concurrency as a program structuring technique is also available to application programs. It is not our intention in this book to deal with operating systems as a subject, but it will inevitably be the case that operating systems will provide a fertile source of examples of concurrent programs.

1.3 Concurrency in Programs

Any sequential program is in all probability not necessarily totally sequential. That is, it is often the case that the statements of the program could be re-ordered to some extent without affecting the behaviour of the program. It is usually possible, however, to identify those parts of the program (it is useful to think in terms of program statements in your favourite high-level programming language) which do depend on one another, in the sense that one statement must

precede another. A simple example would be the case where a program might include the statement

$$x := x + 1;$$

This statement should really only be used if there has been a previous statement initialising the variable x, e.g.

$$x := 0;$$

We could examine any sequential program and identify all such dependencies, and we would almost certainly find that there were quite a number of pairs of statements for which no such dependencies exist. In other words, it is possible to identify a partial ordering of program statements which define the interdependencies within a program.

Suppose, for example, that a program consisted of statements denoted by the letters $A, B, C, D, E, F, G, H, I$ and J. Suppose also that we were able to write down the partial ordering as a set of relations:

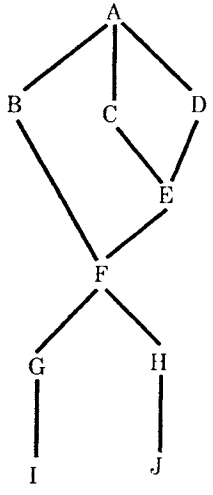
$$A < B, A < C, A < D, C < E, D < E, B < F,$$
$$D < F, E < F, F < G, F < H, G < I, H < J$$

where the relational operator $<$ is meant to be interpreted as "must precede in time". The first property of this partial ordering we notice is that the relation $D < F$ is in fact unnecessary, since it is a consequence of the two relations $D < E$ and $E < F$, this ordering relation having the transitivity property. The partial ordering defined by these relations may be illustrated by a directed graph as shown in figure 1.2.

1.4 An Informal Definition of a Process

It is assumed that any programmer who attempts to write concurrent programs will have had a reasonable amount of experience of conventional sequential programming. With this in mind, we attempt to decompose our concurrent programming problem into a set of sequential programs together with some controlled interaction between them. Thus we put forward as the basic building block of a concurrent program the *sequential process* (where no confusion can

Figure 1.2



result, we shall abbreviate this to *process*). Perhaps the simplest, and also the least useful definition of a sequential process, is to describe it as the activity performed by a processor. This however begs the question of what is meant by a processor. We have an intuitive notion of what a processor is, namely a device which is capable of performing a sequence of well-defined instructions one at a time. It is largely the case that a sequential process corresponds to an ordinary sequential program, but the use of the word *process* is intended to convey an impression of "activeness" which the word *program* may not. A more formal definition of *process* will be attempted in the following chapter, but for now it will suffice to describe a process as an activation of a program or sub-program. This implies that storage has been made available both for the code being executed and for the data upon which that code operates. It is also useful to think for the moment of the allocated storage space as being distinct from the storage space available to any other process.

Using this notion of the sequential process, we can see that it would be quite possible for two processes to be active at the same time

and to be running the same code, i.e. one program can be associated with two distinct processes simultaneously. Conversely, if two programs were to be run strictly sequentially but making use of the same storage area, we might regard them as belonging to the same sequential process. Thus processes and programs are not the same, although clearly "process" would be a somewhat empty concept without an associated program. In the case where two processes are executing identical code, and that code is *pure*, i.e. is never modified as a result of its own execution, then some machines or systems may permit two (or more) processes to access and execute the code from the same physical storage. This has the virtue of saving on actual storage used, but conceptually one should think of separate processes as being totally disjoint and occupying distinct areas of storage.

The question then arises as to how processes are created and how, if at all, they cease to exist. The simplest approach to this problem is to imagine that processes are brought into existence as the system begins to operate, and to continue to exist until the whole system is halted. Such a model has great virtue in the consideration of an operating system, which typically is required to provide services continuously from the starting up of the system to the halting of the machine. Similarly, a general concurrent program may have a fixed number of processes which are initiated as soon as the program starts to run, and remain in existence until the whole program terminates.

Some programming systems, usually in collaboration with the underlying operating system, are capable of creating and destroying processes dynamically. In such cases, the concurrent program clearly has much more flexibility with regard to the way in which processes may be created and destroyed according to the needs of the whole program, and may affect the overall structure of the program.

1.5 Real Concurrency and Pseudo-Concurrency

It is almost always the case that a system allowing the use of multiple concurrent processes (or its users) will require more processes than the number of physical processors provided by the hardware. In those rare cases where the number of concurrent processes is less than the number of available physical processors, we shall refer to the concurrency as *real* or *true* concurrency.

In the more usual situation where the program (or system) requires more processes than there are processors available, in order not to restrict the system arbitrarily, it is necessary that some mechanism be provided which will simulate the action of a number of processes using a single processor only. This may be achieved by running the processor under the control of a (preferably small) program commonly known as a *kernel*. The responsibilities of the kernel will vary from system to system, but it must provide the abstraction of multiple concurrent processes. This is usually implemented using some kind of *time division multiplexing* of the processor. Concurrency provided in this way will be referred to as *pseudo-concurrency*. A kernel which provides only the multiple-processor abstraction is actually not very useful, and so in general a kernel will also offer some form of inter-process communication mechanism, and possibly some primitive operations to allow the dynamic creation and deletion of processes. We shall discuss the implementation of a kernel in a later chapter, and it is sufficient to note now that provided the kernel is doing its job correctly, pseudo-concurrency and true concurrency will be indistinguishable at the concurrent programming level.

Since we assume that real concurrency and pseudo-concurrency are indistinguishable at this level, the reader may be wondering why the two terms have been introduced. In fact, it is sometimes assumed that the fact that there is only one processor being shared amongst all the processes makes the problems of controlling the interaction easier. That may be so in some instances, but we hope to show that the simple method of preventing interference, namely preventing the processor from being multiplexed, is not the best solution in many cases, and that by regarding pseudo-concurrency as being indistinguishable from real concurrency, general, and usually more satisfactory solutions to the problems of concurrent programming can result.

1.6 A Short History of Concurrent Programming

It is often assumed that Dijkstra instigated the study of concurrent programming in his now classic article "Co-operating Sequential Processes", published in 1967. Certainly in that article we see the introduction of some now well-known problems such as "The

Dining Philosophers", "The Sleeping Barber" and "The Dutch Flag Problem", and perhaps most importantly, the *critical section problem* and its solution using *semaphores*. This article was indeed the first to take a "high-level" view of concurrent programming. As an aside, it is interesting to note that the same article introduces the notion of *deadlock*, and presents an algorithm which can detect the possible presence of deadlocks.

However, a more machine-language-oriented approach to concurrency was presented in the early 1960's, first by Conway and then by Dennis and Van Horn. The idea of multiple threads of control was introduced at this time, and it is interesting to observe that some of the problems of accessing shared resources such as memory were also addressed at that time.

As in the development of high-level (sequential) programming languages, researchers came to realise the problems associated with using these low-level constructs, mainly in the area of trying to write programs correctly and quickly, and as a result higher-level and more restrictive constructs were invented. Thus we now see a plethora of different techniques for providing controlled concurrent programming, many of which we shall be examining in greater detail in the later chapters of this book.

1.7 A Map of the Book

We shall examine the fundamental structures of concurrent programming in the following chapter, showing how concurrent activity may be specified and introducing the principal components from which concurrent programs are constructed. This will include a more formal definition of the notion of a process than that given earlier in this chapter, and we shall also examine more closely the ways in which processes may be created and destroyed. In chapter 3, we shall deal in greater depth with the problems of interaction between processes, showing how control of such interactions can be achieved with only the minimum of assistance from the hardware, while at the same time demonstrating that some hardware assistance is necessary. From these small beginnings, we shall build up a whole hierarchy of concurrency control techniques. At an appropriate stage, we shall call for help on slightly more functionality in the hardware. This chapter deals with the lowest level of interaction which is concerned with the

non-interference of processes at critical points in their respective executions. At a higher level, we shall be interested not only in the problem of non-interference of processes, but also in the positive interaction between them. A number of high level constructs for communication between concurrent processes have been proposed, and these will be dealt with in chapters 4 and 5.

In describing these higher-level programming structures for the control of concurrency, we will show how such structures provide more natural ways in which to express the solutions to concurrency problems, in much the same way as high-level programming languages provide a more natural way of solving ordinary sequential programming problems than can be achieved using machine or assembly language. It is convenient to classify the type of interaction which takes place between concurrent processes into two classes; shared data and message passing. In the former case, a process is permitted to access data which is also accessible by one or more simultaneously active processes. We shall see that difficulties can be introduced by unconstrained simultaneous access to shared data, and chapter 4 discusses methods by which such problems can be avoided. In chapter 5, we examine various methods and mechanisms for passing information between processes without the necessity for allowing explicit access to a shared resource. In chapter 6, we illustrate some of the principles by discussing various solutions to a particular problem in concurrent programming, using a number of concurrent languages which are presently available.

Finally, in chapter 7, we give some indications of how some of the concurrency structures introduced in the earlier chapters may be implemented, including the construction of a multiplexing kernel to provide the illusion of multiple processors.

1.8 Exercises

1.1 Draw the tree which represents the evaluation of the arithmetic expression:

$$(a ** 2 + (c + d) * e) + (f + g * h)$$

How many operations does this evaluation require?

1.2 Assuming that access to a variable or constant is instantaneous, and that each operation takes one unit of time, how many units of time would be required to evaluate the expression given in exercise 1.1

(a). sequentially,
(b). concurrently, as defined by the tree structure (assuming that sufficient processors are available to allow maximal concurrency).

1.3 Repeat exercise 1.2, given that the time taken by each operation is:

\+ - 1 unit
* - 2 units
** - 4 units.

1.4 Given the following Pascal program:

```
        program SimpleArithmetic;
        var a, b, sum, diff : integer;
        begin
(A)         a := 25;
(B)         b := 17;
(C)         sum := a + b;
(D)         diff := a - b;
        end; { SimpleArithmetic }
```

construct a partial ordering on the statements of the program (labelled *A*, *B*, *C* and *D* at the left hand side of each executable statement).

1.5 For the program given in exercise 1.4, and using the partial ordering found, draw the corresponding acyclic graph.

1.6 In the following Pascal program, label the various iterations of the **for** statements in a suitable way (for example, by the letters $A_1, A_2, \ldots A_5, D_1, D_2, \ldots D_5$)and hence show the extent to which this program may also be made concurrent:

```
        program Fibonacci;
        var i : 1..5;
            A : array [1..5] of integer;
            F : array [0..6] of integer;
        begin
(A)         for i := 1 to 5 do A[i] := 0;
(B)         F[0] := 0;
(C)         F[1] := 1;
(D)         for i := 1 to 5 do F[i+1] := F[i-1] + F[i];
        end; { Fibonacci }
```

1.7 Examine the component actions in an everyday task such as preparing a meal. How much concurrency is there already? To what extent could additional concurrency be introduced if there were two (or more) people involved instead of one? Where do conflicts arise? What constraints are there which prohibit the introduction of concurrent activity?